# Efficient Data Redistribution to Speedup Big Data Analytics in Large Systems

Long Cheng[1] and Tao Li[2]
[1]Department of Mathematics and Computer Science, Eindhoven University of Technology, The Netherlands
[2]Faculty of Computer Science, TU Dresden, Germany
l.cheng@tue.nl tao.li@tu-dresden.de

***Abstract*—The performance of parallel data analytics systems becomes increasingly important with the rise of Big Data. An essential operation in such environment is parallel join, which always incurs significant cost on network communication. State-of-the-art approaches have achieved performance improvements over conventional implementations through minimizing network traffic or communication time. However, these approaches still face performance issues in the presence of big data and/or large-scale systems, due to their heavy overhead of data redistribution scheduling. In this paper, we propose *near-join*, a network-aware redistribution approach targeting to efficiently reduce both network traffic and communication time of join executions. Particularly, near-join is lightweight and adaptable to processing large datasets over large systems. We present the details of our algorithm and its implementation. The experiments performed on a cluster of up to 400 nodes and datasets of about 100GB have demonstrated that our scheduling algorithm is much faster than the state-of-the-art methods. Moreover, our join implementation can also achieve speedups over the conventional approaches.**

***Keywords*-data analytics; parallel joins; data locality; data-intensive computing; high performance computing**

## I. Introduction

With growing of Big Data, a large quantity of data-intensive, high-performance computing applications are challenging current data processing algorithms and systems. One of the most common operations in data analytic applications is parallel join. This operation facilitates combination of two relations based on a common key, and always incurs heavy cost on network communication within a distributed system. As reported in [1], expensive queries could spend more than 65% of their completion time on transferring tuples over networks. Improving efficiency of data transferring will bring in significant performance enhancement for big data analytic applications.

A typical parallel join implementation can be, in general, decomposed into an initial redistribution stage followed by a local join process. As the latter process has been extensively studied [2] and its cost does not contain any inter-machine communication, for the purpose of this work, we mainly focus on the redistribution process over a distributed system[1]. For a single join operation between two input relations $R$ and $S$ over a system consisting of $n$ nodes, we assume that tuples in $R$ and $S$ are in the form of <key, payload> and $|R| < |S|$ in the following.

[1]For local join analysis, we have selected the commonly used hash join (i.e., hash table building & probing) approach in this work.

The *hash redistribution* approach [3] is a widely used method (we use the term hash-join in the following) for parallel join executions. In hash-join, the initially partitioned relation $R_i$ and $S_i$ on each node $i$ are firstly partitioned into distinct sets $R_{ik}$ and $S_{ik}$ respectively, based on hash values of their join key attributes. Then, each of these sets is distributed to remote nodes for final local processing [4]. An example of such data redistribution scheme is shown in Figure 1. There, the tuples with the key 3 (assuming its hash value is 3) in $R$ and $S$ will be moved to node 3 to complete local join. If we quantify the cost of network communication by the number of tuples moved to **remote** nodes[2], then the cost in this example will be 11.

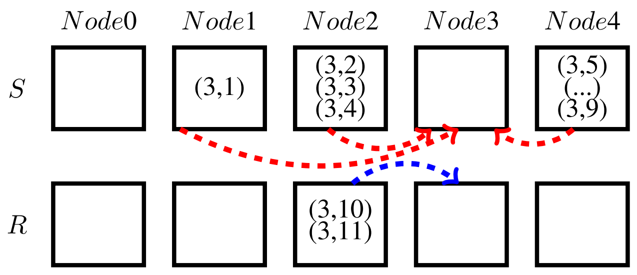

Figure 1. An example of data redistribution in a hash join.

The hash-join is far from optimal network cost due to transmission of almost all input tuples. To overcome this inefficiency, various approaches have been proposed and one of them is *partial duplication* [3]. In this scheme, a part of tuples in the set of small relation are broadcast to all nodes. For instance, the two tuples of $R$ at node 2 in the above example will be broadcast so that network cost can be reduced from 11 to 8 compared to the hash-join implementation. Because of its efficiency, the approach partial duplication [3] has been adopted by many commercial systems (e.g., Teradata [3], Microsoft [5] and Oracle [6]).

Compared to hash-join and partial duplication, the state-of-the-art *track-join* [1] is able to minimize network traffic by employing a more fine-grained method based on data

[2]Actually, the cost should be quantified by the data volume of transferred tuples. For simplicity, we assume the size of each tuple remains same for both algorithm design and experiments.

locality assignment. To reduce communication time, another approach named neo-join [7] uses techniques of communication scheduling to avoid network congestion in data redistribution. These two approaches have been shown to be efficient on join executions and outperform the conventional hash-join. However, as we will discuss later in Section II, both methods still face performance issues in the presence of big data and/or large data systems because of their heavy scheduling overhead for data redistribution, .

In this paper, we present an efficient algorithm for data redistribution, with targets to reduce both network traffic and communication time in parallel join operations. We have provided the detailed implementation of our approach and conducted a performance evaluation using 100GB data over up to 400 computing nodes. We summarize the contributions of this work as following:

- We introduce the state-of-the-art data redistribution approaches in parallel joins and analyze their possible performance issues in processing big datasets on large-scale systems.
- We present a new approach named *near-join*, a **ne**twork-**a**ware **r**edistribution approach for efficient data transferring by exploring data partitioning and locality assignment.
- We compare near-join with state-of-the-art techniques and show that our approach is lightweight and much faster to process big data on large systems. Additionally, our algorithm is shown to significantly outperform hash-join in the aspect of join execution time.

The rest of this paper is organized as following. In Section II, we introduce the state-of-the-art techniques and discuss their performance issues. We present our near-join algorithm and its implementation details in Section III. We carry out extensive evaluation of our approach in Section IV. We report the related work in Section V and conclude this paper in Section VI.

## II. Background

In this section, we first introduce two state-of-the-art techniques, track-join and neo-join, which aim to minimize network traffic and communication time. Then, we discuss their potential performance issues in the presence of large datasets and large systems.

### A. Optimization on Network Traffic

Large amount of data movement operations always consume tremendous network resources and result in long communication time. Thus, reducing the volume of transferred data over networks would bring in performance improvement on join executions. Compared to hash-join and partial duplication approaches, track-join [1] uses a more advanced approach that can reduce network traffic.

Based on the complete knowledge of location and frequency of each join key, the analysis on network cost for a key in track-join is generally composed of two operations. (1) *Select broadcast:* tuples in a relation are selected to broadcast to all other nodes, except of the nodes with no matching key. In Figure 1, the two tuples in $R$ at node 2 will be only broadcast to the node 1, 2 and 4, and the network traffic in this case is 4. (2) *Migration:* for the other relation, tuples on each node will be migrated to another node or just stay in their place, depending on whether the overall network cost is reduced. For instance, starting with the tuple <3,1> at node 1, if this tuple is moved to node 1, the communication cost will increase to 7. Therefore, another migration to node 2 will be probed. In this case, the communication cost is reduced from 4 to 3 (i.e., the cost is 4+1-2, since the two tuples in $R$ do not need to move to node 2 anymore), thus the tuple will be moved to node 2. Following by this, the analysis on migration of the data at node 3 (note that there will be four tuples on this node now) is commenced and the whole process terminates until the data migration at the last node $n$ finishes.

It can be seen that the implementation of track-join is very efficient on reducing communication cost. The reason is that it adopts a backward way to search all possible opportunities on reducing network traffic. Knowing the destination(s) of all keys, input tuples will be transferred based on the redistribution plan during join executions. Actually, track-join can be considered as an approach which extensively uses the philosophy of moving *small* data chunks instead of *large* data chunks in a join implementation.

### B. Optimization on Network Communication Time

*Network congestion.* In a distributed system, reducing network traffic can significantly reduce communication time. However, minimizing the communication traffic does not necessarily lead to minimal communication time. The reason is that when computing nodes use the network without any coordination, utilization of network bandwidth could be very poor. For example, in a common hash-join implementation, all nodes firstly send their tuples to the first node, then to the second node, and so on. This would lead to significant network congestion, because the nodes compete for the bandwidth of a single link while other links are not fully utilized [7].

*Communication scheduling.* Dividing network communication between nodes into distinct phases is an efficient way to avoid network congestion. In such a scheme, for each phase, a node has a single destination to send data, and likewise a single source to receive data. It is very similar to a parallel peer-to-peer communication pattern in networks. The *round-robin* scheduling is one of such approaches, in which each node sends a data chunk to its neighboring node in each phase. However, the sizes of transferred data among nodes could be different in each phase, thus this approach still can not fully utilize link bandwidth. In order to maximize bandwidth utilization and consequently reduce communication time, we need a schedule on data transferring among nodes in each phase.

Actually, the above problem has been studied as the *open-shop* scheduling problem [7], [8], and the optimal solution always guarantees that the communication time with duration $t$ can be achieved, where $t$ is the maximum value of the sum of data sending/receiving time in each phase for each node. If the bandwidth between each pair of node is same[3], then the time $t$ can be represented by number of received tuples at a node with the maximum received tuples. For example, with the track-join implementation, the number of sent/received tuples at the five nodes in Figure 1 is {0,1,2,0,0}/{0,0,1,0,2}. Then, the minimal network communication time will be 2, because 2 is the maximum value in this case.

*Minimize communication time.* We denote the data transferring (i.e., sending/receiving) cost at each node $i$ as $t_i$. To achieve minimal communication time, our target is to *minimize* the following expression, the value of which highly relies on the locality assignment (i.e., destinations) of the initially partitioned data (e.g., hash partitioned) at each node.

$$max\{t_i\} = max\{s_i, r_i\} \qquad \forall i \in [0, n) \tag{1}$$

The same cost model in [7] is used: each node has $p$ data partitions, the size of partition $j$ at node $i$ is denoted as $h_{ij}$ and the decision variables $x_{ij} \in \{0, 1\}$ are used to indicate whether a partition $j$ at each node is assigned to node $i$. Namely, $x_{ij} = 1$ represents that partition $j$ is assigned to node $i$, and $x_{ij} = 0$ means not. The constraint that each partition is assigned to only one node should be fulfilled and we compute the receiving cost $r_i$ as following[4]:

$$\sum_{i=0}^{n} x_{ij} = 1 \qquad j \in [0, p) \tag{2}$$

and

$$r_i = \sum_{j=0}^{p-1} \big(x_{ij} \sum_{k=0, i\neq k}^{n-1} h_{kj}\big) \qquad i \in [0, n) \tag{3}$$

As presented in [7], the above cost analysis on data assignment can be reformulated as a *mixed integer linear programming* problem and an optimal solution can be computed by using an optimizer (e.g., Gurobi[5]). With the open-shop scheduling, the network communication time of a join execution can be efficiently reduced.

[3]The method can be extended to the case with different bandwidth, regardless, this is out the scope of this work. Also, we do not enforce it in our experiments.

[4]Note that the analysis on the cost of sent tuple $s_i$ is very similar as the received tuples $r_i$, for the sake of the simplification of our presentation in this paper, we just focus on $r_i$ in our discussion and implementation in the following. Namely, when we refer to $r_i$, we actually mean the transferring cost $r_i$ as we have described.

[5]www.gurobi.com

### C. Discussion

The experimental results presented in [1] and [7] show that track-join and neo-join can efficiently speedup join executions compared to hash-join. However, both approaches would have performance bottlenecks for processing big data on large-scale systems.

The reason is that the overhead of scheduling in these two approaches could be heavy: (1) track-join uses four-phase scheduling to track the locality of all join keys. The first phase is done by a pre-join of all the unique keys, which is expensive when the number of unique keys is huge. Moreover, differently from a simple local join, the pre-join process has to build a more complex hash table at each node, which is in the form of (key, (list[N_r], list[N_s])) to record the key's frequency and location at each node, and this could bring in more overheads; and (2) the data locality assignment in neo-join relies on solving NP-complete optimization problems and its solving time will increase sharply with increasing the number of computing nodes (and also the number of partitions). Such computational complexity makes this approach not suitable for nowadays large data systems with hundreds or thousands of nodes.

Additionally, track-join focuses on reducing the whole network traffic in a distributed environment. It neglects the fact that the overall transmission time actually depends on completion time of each node. On the other hand, patterns of join operations used in neo-join are still based on the conventional redistribution and broadcasting approaches lacking a fine-grained data movement control such as *select broadcast* as we have described[6].

In comparison, as we will show later that our near-join considers both the network traffic volume and network transmission time. Most importantly, we keep our schedule process lightweight for processing big data on large data systems.

## III. Our Approach

In this section, we present the design of the near-join algorithm and its implementation details. Additionally, we compare near-join with multiple techniques, including two state-of-the-art approaches, to show its advantages.

### A. Overview

To achieve low overheads on the scheduling process, we follow two basic design principles for our near-join: (1) network traffic minimization techniques should be only applied to part of the data that can greatly reduce network traffic instead of the whole input set, so as to make the

[6]Note that the authors of neo-join also use the term *select broadcast* in their paper, regardless, they actually mean the *partial duplication* approach as described in Section I. The reason is that the former operation only broadcasts data to some specified nodes and the latter one broadcasts to all the nodes. The *partial* just means part data of the full relation but rather than part of the nodes in current literature [3], [4].

approach applicable to big datasets; and (2) more practical and simple approach should be applied to reduce the value of $max\{r_i\}$ during data assignment, so as to efficiently reduce network communication time and also make the method applicable to large systems.

Based on these two design principles, the main process of our near-join implementation can be divided into the following three main phases:

- *Data partitioning:* Input tuples in both $R$ and $S$ are partitioned into two groups, skew ones and non-skew ones.
- *Schedule processing:* This phase contains the following three steps:
  - *step 1.* Use track-join [1] to maximize the data locality for the skewed tuples to minimize the network traffic in this step.
  - *step 2.* Based on data volume received at each node in *step 1*, use an optimized approach (e.g., neo-join [7]) to redistribute the rest non-skew tuples, so as to reduce the final $max\{r_i\}$.
  - *step 3.* Based on the number of transferred tuples between all the nodes (i.e., an $n \times n$ matrix) in the above two steps, use the open-shop solution [8] to schedule the data transfer among nodes to guarantee that the minimal network communication time can be achieved.
- *Join implementation:* Tuples in both $R$ and $S$ will be redistributed based on the transfer plan created in the second phase. The local join implementation will commence once the redistribution is done and the entire join process terminates when all individual computation at nodes finishes.

Our method is different from current approaches: we consider both network traffic and network communication time (i.e., the value of $max\{r_i\}$) in our join implementation. We refer our approach as *network-aware redistribution*, because both the network traffic and communication time are *trackable* during our data locality assignment process.

## B. Implementation of near-join Approach

In our join implementation, the skew detection (e.g., use sampling) and local join implementation in phases of data partition and join implementation, have been extensively studied in literature. In the meantime, the implementation details of track-join and open-shop scheduling in the step 1 and 3 of the second phase, have been introduced by [1] and [7] respectively. Specifically, with the track-join, we can process all the skewed keys in parallel and get the corresponding *K-($N_1$,$list[N_2]$)* mappings for all the skewed tuples in both $R$ and $S$. Here, $K$ stands for a skewed key appears at node $N_1$, and $list[N_2]$ is a list of destination nodes that $K$ is assigned to. In such scenarios, we only focus on the implementation details of the *step 2* in the second phase in our near-join approach to show how to efficiently assign the locality for non-skew tuples.

*1) Non-skew data assignment:* We assume that the number of received tuples at each node $i$ is $r_{i0}$ after processing by track-join (i.e., the *step 1*). If we want to get optimal data assignment for non-skew tuples, one solution is to use the analysis in neo-join. Then, the Eq. (3) in Section II can be rewritten as the Eq. (4) as below.

$$r_i = \sum_{j=0}^{p-1} \big( x_{ij} \sum_{k=0, i \neq k}^{n-1} h_{kj} \big) + r_{i0} \qquad i \in [0, n) \tag{4}$$

Similar to neo-join, solving the optimization problem based on Eq. (4) will be still costly when the number of node is large. The reason is that each $r_{i0}$ will be a constant value and the complexity of the problem will be still NP-complete. To address this issue, we proposed a more practical method as following to reduce the final $max\{r_i\}$ in the data assignment process.

For the initial (hash) partitioned non-skewed data, the tuples in both relations, $R$ and $S$, with the same hash value must be assigned to the same node to implement the local joins. Namely, our target is to get the data assignment results, which can be represented as the *H-N* mappings. Here, each *H-N* is a mapping between the hash value of a key and the node that the key is assigned to. Based on this, we use a *heuristic* way to assign the node *N* for each *H* step by step. As shown in Algorithm 1, our approach contains three main steps. For simplicity, we assume that each $r_{i0} = 0$ (lines 2-3) at the beginning of our algorithm.

- *step 1.* Get the statistical information of hash values at each node $(h, node, num)$, where $h$ is a hash value for the hash-partitioned tuples, $node$ is the location information of $h$ and $num$ is the number of tuples with the hash value $h$. This step can be considered as the pre-processing (line 1) and can be done in parallel at each node.
- *step 2.* The $(h, node, num)$ information at each node will be collected and combined by a master node (line 4), which is responsible to compute the data assignment. After that, as presented in line 5, a two-level sorting method is applied to the list $\mathcal{L}$ based on the maximum value of $num$ in different levels.
- *step 3.* The *H-N* solution is computed over the sorted data in a sequential order (lines 6-9). For each hash value $h$, we track the $max\{r_i\}$ (i.e., $R_{max}$) for all possible destinations (i.e., in total $n$ possibilities as lines 11-23) and choose the node achieving minimal $max\{r_i\}$ (line 24). Then the received tuples at each node $R$ will be updated for computation of the next hash value.

It can be observed that our implementation is very similar to a step-by-step balancing approach. For the data assignment of each hash value, we always try to balance the

**Algorithm 1** Non-skew data processing in near-join

```
    Pre-processing:
 1: We can easily get the $(h, node, num)$ information at each node
    once the input relations have been partitioned (i.e., using hash
    or range etc.).

    Initialization:
 2: $R = [r_0, r_1, ..., r_n] = 0$    // initially received data
 3: $R_{max} = 0$    //monitor the possible runtime

    Main procedure:
 4: Collect all the $(h, node, num)$ at each node and combine
    them based on the value of $h$, and then we have that $\mathcal{L} =$
    $list\big(h, list(node, num)\big)$.
 5: Sort $\mathcal{L}$ with descending order in two levels: (1) for each $\mathcal{L}_i \in$
    $\mathcal{L}$, sort $\mathcal{L}_i$ based on the maximum value of $num$ in the list
    $L = list(node, num)$; and (2) for each $L$, we sort each pair
    $(node, num)$ based on the value of $num$. Additionally, for
    each $L$, We also calculate the sum of each $num$, referred as
    $\mathcal{N}$.

 6: for each $\mathcal{L}_i \in \mathcal{L}$ do    //sequentially access each $\mathcal{L}_i$
 7:     $\mathcal{P}_i$(H-N) = compute($\mathcal{L}_i$, $R$, $R_{max}$)    //compute H-N
 8:     Add $\mathcal{P}_i$ mapping in an array $\mathcal{P}$
 9: end for
10: return The H-N mappings $\mathcal{P}$    //get all the H-N pairs

    Procedure compute($\mathcal{L}_i$, $R$, $R_{max}$):
11: for ($j = 0, 1, ..., n-1$) do    //loop over all the nodes
12:     H$= \mathcal{L}_i.h$, N$= \mathcal{L}_i.L_j.node$    //targeted node
13:     $S_{trans} = \mathcal{N} - \mathcal{L}_i.L_j.num$    //the transferred data
14:     $R_j = max\{R_{max}, R[N] + S_{trans}\}$
15:     if $j = 0$ then    //the first condition
16:         if $R_0 = R_{max}$ then
17:             $R[N] += S_{trans}$    //update received
18:             return $\mathcal{P}_i$(H-N)
19:         end if
20:     else if $R_j < R_{j-1}$ then
21:         $R_{max} = R_j$    //record the minimum solution
22:     end if
23: end for
24: Track the loop $x$ in line 11 with achieving $R_{max}$
25: N$= \mathcal{L}_i.L_x.node$    //node information
26: $R[N] += \mathcal{N} - \mathcal{L}_i.L_x.num$    //update $R$
27: return $\mathcal{P}_i$(H-N)
```

number of received tuples at each node, to guarantee that a smallest $max\{r_i\}$ can be achieved. Consequently, the value of the final $max\{r_i\}$ is able to be efficiently reduced.

We use two sorting operations in *step 2*. The main reason here is that the variation of the number of received tuples at each node is more sensitive on large data chunks rather than small ones. The first sorting operation can guarantee that hash values with large number of tuples are processed with higher priority than the small ones. On the other hand, the second sorting operation is used to make sure that searching of a target node can be terminated as early as possible, and also with minimal network traffic (as shown in lines 16-18 in the algorithm). Namely, sometimes we do not need to examine all the possible data movements. For example, for a given $h$, with the sorted list of *(node, num)*, as we examine the $max\{r_i\}$ over each $num$ in a descending order, we can stop searching when the value of $max\{r_i\}$ in current loop is equal to previous one[7]. The reason is that, in such a case, the $max\{r_i\}$ does not increase and increased network traffic ($\mathcal{N} - num$) is the smallest one (because the value of $num$ in the followed searching process will be smaller than current one).

*2) An example:* To give a more intuitive view of our near-join, we show a simple example here. We assume that there are three nodes, each node has 32 tuples and all of them have been partitioned into 8 chunks according to their hash values. With the statistic information $(h, node, num)$ collected from each node, the list $\mathcal{L}$ can be represented as a table in Table I. For instance, the first row in the table represents the item $\big(0, (0, 1), (1, 3), (2, 11)\big)$ in $\mathcal{L}$. Namely, it shows that the number of tuples with hash value 0 on the three nodes is 1, 3 and 11 respectively.

Table I
AN EXAMPLE OF PARTITIONING INFORMATION ON EACH NODE

| hash values/num | $Node0$ | $Node1$ | $Node2$ |
|---|---|---|---|
| 0 | 1 | 3 | 11 |
| 1 | 1 | 10 | 3 |
| 2 | 1 | 10 | 2 |
| 3 | 4 | 3 | 9 |
| 4 | 7 | 1 | 2 |
| 5 | 7 | 2 | 2 |
| 6 | 7 | 2 | 3 |
| 7 | 4 | 1 | 0 |
| sum | 32 | 32 | 32 |

As shown in Table II, after sorting over the maximum $num$ in each item in $\mathcal{L}$, we start to assign data for each hash value. For the first case with hash value 0, we have three options for the target node, namely, we can move all the tuples to either node 0, 1 or 2. Consequently, the number of tuples received from remote nodes will be 14, 12 and 4 respectively. In this case, we select the smallest $max\{r_i\}$ in the three movement options. Since the value 4 is the smallest one, all the tuples with hash value 0 will be assigned to node 2. Namely, the first *H-N* mapping will be *(0-2)*.

Based on the number of received tuples at each node in the first step, for the case with $h = 1$, we also have three options as well. Since moving data to node 1 leads to smallest $max\{r_i\}$, the mapping *(1-1)* will be selected. Actually, we have sorted *(node, num)* based on the value of each *num* in our second-level sorting operation, thus, the cost of assigning all tuples to node 1 will be computed at first. In this case, the value of $max\{r_i\}$ is still 4 and

[7]Note that, with the increased number of received tuples at each node, $max\{r_i\}$ in current loop will be not smaller than previous one.

Table II
DATA ASSIGNMENT PROCESS IN *near*.

| $h$ | $N_0$ | $N_1$ | $N_2$ | $r_0$ | $r_1$ | $r_2$ | $max\{r_i\}$ | *opt.* |
|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 3 | 11 | 14 | 0 | 0 | 14 | |
| | | | | 0 | 12 | 0 | 12 | |
| | | | | 0 | 0 | 4 | 4 | $\surd$ |
| 1 | 1 | 10 | 3 | 13 | 4 | 0 | 13 | |
| | | | | 0 | 4 | 4 | 4 | $\surd$ |
| | | | | 0 | 0 | 15 | 15 | |
| 2 | 1 | 10 | 2 | 12 | 4 | 4 | 12 | |
| | | | | 0 | 7 | 4 | 7 | $\surd$ |
| | | | | 0 | 4 | 15 | 15 | |
| ... | | | | | | | | |

does not increase. Therefore, we can directly terminate our searching in the current step and go to the next hash value. The computing will be terminated until all the hash values have been examined.

### C. Comparison with Current Approaches

If we compare our near-join with the conventional hash-join, there are two main advantages of our approach: (1) network traffic can be highly reduced. The reason is that we have explored data locality in our implementation. By using the techniques from track-join, the cost on transferring large number of skewed tuples can be minimized. In the meantime, for the non-skew data, we also guarantee that the network traffic is reduced by considering possible communication time. In contrast, hash-join does not consider data locality and simply redistribute all the input tuples; and (2) network communication time can be efficiently reduced. This is not only caused by the reduction of the network traffic but also that we have used a tag $max\{r_i\}$ to track the possible communication time during data assignment and thus guarantee that a near optimal solution can be achieved. Moreover, with the open-shop scheduling, we can also efficiently improve the utilization of network bandwidth.

Compared to the two state-of-art approaches, track-join [1] and neo-join [7], our near-join has provided an efficient way to remedy their main performance issue as we have discussed in Section II-C. Our scheduling process is lightweight and applicable to process big data on large data systems. The reason is that our data assignment method does not focus on examining locality for all the join keys or solving NP-complete problems. In comparison, we only conduct a *per-key* statistics for the skewed keys, the number of which is always very small. In the meantime, for non-skew keys (tuples), we only search possible destinations for each hash partitioned data, the complexity[8] of which is $O(n^2)$. Another advantage is that we have taken advantages of both two approaches/ideas in our algorithm. We try to reduce **both** network traffic and communication time. This could make our join executions have heavier network traffic than track-join as well as perform slower than neo-join on network communication. However, as we will show in our later experiments in Section IV-C, the impacts of these differences are neglectable in terms of general communication time in a join, compared to the improvements we achieved on reducing the complexity of schedule process. In such scenarios, our approach can be considered as an efficient and practical solution which bridges the gap between the two state-of-art techniques.

[8]Note that the complexity is actually $O(n \cdot p)$, and $p$ can be represented by $c \cdot n$, where $c$ is greater than 1 and is a constant.

## IV. EXPERIMENTAL EVALUATION

In this section, we present an experimental evaluation of our algorithm and compare it with current approaches.

### A. Platform and Setup

Our evaluation platform is the HRSK-II system of ZIH at TU Dresden. Each node we used has two 12-core Intel Xeon CPU E2680 processors running at 2.50 GHz, resulting in a total of 24 cores per physical node. Each node has about 60GB of RAM and a single 128GB SSD local disk. Nodes are connected by Infiniband. The operating system is Linux kernel version 2.6.32-279 with gcc version 4.4.7.

We implemented hash-join, track-join [1] and our approach with the X10 parallel programming language [9]. The version we used is 2.3 compiled to C++. For neo-join [7], we used the Gurobi version 6.5.1 with C++ to solve the data assignment optimization problem as we have described. To focus on analyzing the major performance metrics of our approach over large-scale distributed systems rather than computing with multiple thread parallelism, we choose 400 physical nodes from our system and only use one core per node. For the hash partition process in our approach and neo-join, without sacrificing generality, we just use a simple hash function $f(k) = k\%p$ and set $p$ to a value which is 15 times the number of used nodes in each test[9]. Moreover, to conduct a fair performance comparison and reduce the impact of I/O, data are always in-memory processed in our tests.

### B. Datasets

We have used the widely used TPC-H benchmark [10] in our tests. We use the following query in our experiments.

```
select *
from CUSTOMER C join ORDER O
on C.CUSTKEY=O.CUSTKEY
```

[9]Note that $p$ is the number of hash partitions. Increase its value would let us be able to have a more fine-grained control on data assignment. However, the costs on scheduling of these two approaches will also increase. Specially, the cost of neo-join increases very sharply in our initial tests. To make part of its results can be presented in this work, we just choose the number 15 here.

The scaling factor of TPC-H is set to 600. The number of tuples in the two input relations is 90 millions and 900 millions correspondingly. In the meantime, the payload in each tuple is set to 100 Bytes, leading to around 100GB input size.

The generated data can be considered as a *uniform* distributed dataset (refer to *skew=0* in the following). As data skew occurs naturally in various applications, in order to control the skewness in our tests, similar to work [3], [11], we randomly choose a portion of data and change their `custkey` to a specified value. For example, we randomly choose 10% of the tuples and set their key to 1, which will make the skewness to 10%. In this way, we can easily identify cheeringly on-going experiment and capture the essence of a Zipfian distribution [12].

### C. Efficiency of Scheduling

We compare the efficiency of our scheduling approach with hash-join, neo-join and track-join. We measure their performance using three metrics defined as following.

- *scheduling time:* The time spent on scheduling for data redistribution in a join. This metric gives insight into the overheads of a join execution.
- *data locality:* The percentage of data that do not need to be transferred during join executions. This metric is an indicator of the volume of network traffic. Higher data locality potentially indicates lighter traffic loads on a network.
- *maximum number of transferred tuples:* The value of the maximum number of sent/received tuples for a node in the system. This metric represents network communication time in a join execution (under the condition that with an efficient communication schedule).

In the following, we report the results of each algorithm over different number of nodes and different data skews respectively.

*1) Over various nodes:* We execute each approach by varying the number of underlying nodes, from 50 nodes (50 cores) up to 400, over the uniformly distributed data. The results for the scheduling time are presented in Figure 2. As hash-join just simply redistributes all the tuples and does not need any scheduling, its cost is always 0. Moreover, for our near-join approach, it can been that it is very lightweight and can always be done within seconds. More importantly, with the increasing number of nodes, its time cost increases very slightly, implying that it can be applied to large systems. In comparison, the two state-of-art approaches perform worse than our approach. For track-join, when the number of nodes is 50, it perform much slower than our approach, and it costs around 115 secs. Though its scheduling time decreases with increasing number of nodes, the decreasing trend becomes not obvious and the time cost is still much greater than our approach when the number of nodes reaches 400. In the meantime, it should be noted that each key of the generated

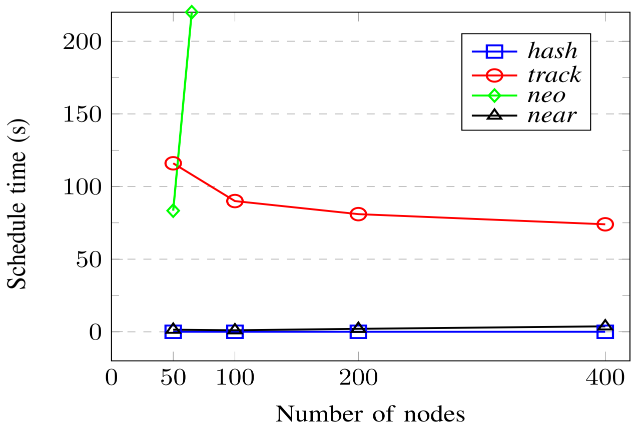

Figure 2. Scheduling time of each algorithm over different number of nodes (with skew=0).

tuples in the relation *order* always appears several times, meaning that the scheduling time of track-join will be even greater if we process more uniform distributed inputs or use larger datasets, due to the increasing number of unique keys. The time cost of neo-join is large and increases sharply with node numbers. For example, it costs 2072 secs for 100 nodes, which is even longer than a regular join execution. This means that neo-join cannot be applied to large systems.

The results of the second and third metric are reported in Figure 3 and Figure 4 respectively. In both figures, we find that hash-join performs only slightly worse than neo-join and our near-join, which is surprising, because the latter two methods have explored advanced techniques on data locality and have considered the maximum number of transferred tuples in their implementations. The possible reason for this could be the number of data partitions $p$ is not big enough or the method we used for data partitioning should be improved. Moreover, it can be seen track-join achieves very high data locality (more than 50%) under different conditions. There are two main reasons for this: (1) the migration and selecting broadcast operation are very efficient on reducing network traffic; and (2) around one third of tuples in the relation *customer* do not have any match in *order* [10], and track-join can avoid the transfer of such tuples through its pre-join process. Though its data locality is much higher than other three algorithms, it can be observed that track-join does not show such a great improvement on the maximum number of transferred tuples. The possible reason for this is that the algorithm only focuses on reducing the whole network traffic but not in a *per-node* way.

*2) Over various skews:* We examine the efficiency of each algorithm over 200 nodes with various skews, increasing from 0 to 20. As neo-join spends more than 5000 secs on its scheduling, which is not meaningful in terms of efficient join executions, we just present the results for the rest three approaches.

As shown in Figure 5, the scheduling time of each

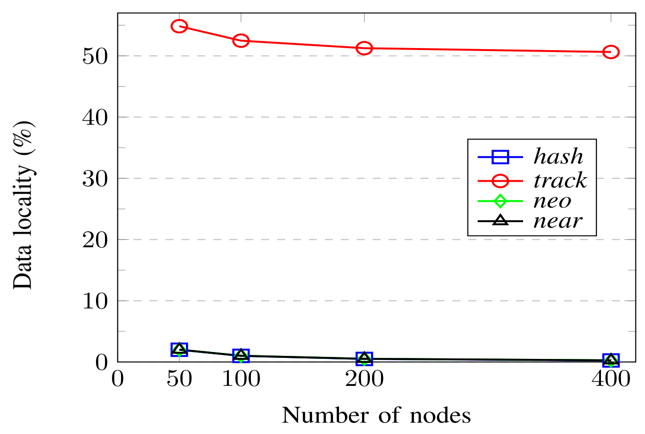

Figure 3. The percentage of non-transferred data for each algorithm over different number of nodes (with skew=0).

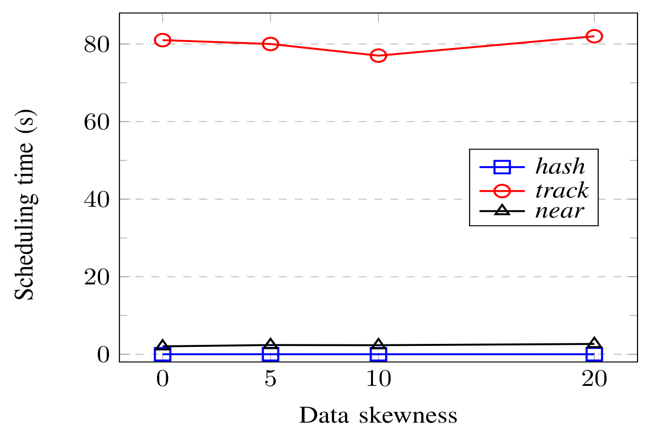

Figure 5. The scheduling time of each algorithm over different skews (over 200 nodes).

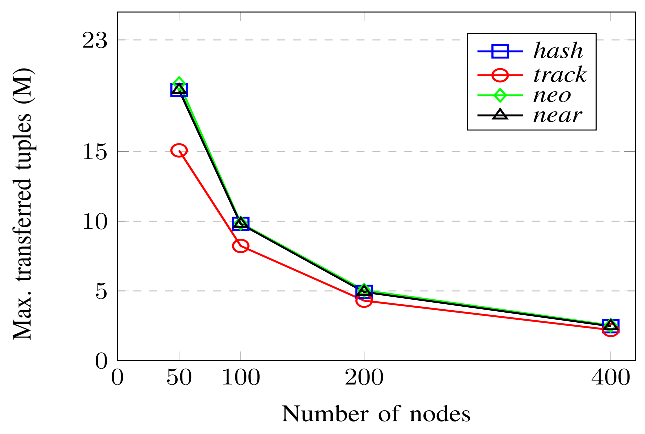

Figure 4. The maximum number of received tuples for each algorithm over different number of nodes (with skew=0).

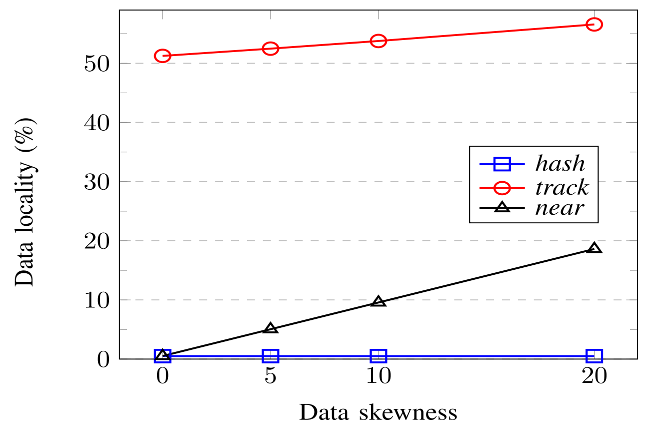

Figure 6. The percentage of non-transferred data for each algorithm over different skews (over 200 nodes).

algorithm is nearly constant when increasing the skew. The reason is that the number of unique keys of input relations does not change for track-join. For our approach, the number of unique skewed keys and the number of hash partitions are constant under different skews. Regardlessly, it can be seen that track-join is still costly under various conditions, compared to our approach. Figure 6 shows that data locality of each algorithm. It can be observed that the data locality of hash-join is always closed to 0 while that of the track-join and our method increases with increasing data skews. The main reasons are: (1) for hash-join, all the tuples still need to be redistributed with increasing skews; and (2) newly generated skewed tuples will be kept locally in our approach as well as in track-join. We can also see that the trend to increase for our approach is more obvious than track-join. The reason is that track-join can get ride of the redistribution of tuples that do not take participate in a join as we have described. With increasing data skew, part of the non-skew tuples, which do not participate in the join, have been changed to skewed tuples, which will be still locally kept and thus make no contribution to increasing data locality.

Though track-join has much higher data locality than our approach, as shown in Figure 7, their maximum number of transferred tuples are nearly same. This implies that their communication time could be same during data redistribution. On the contrary, the communication time of hash-join increases sharply when increasing the data skews, which means this approach could result in more time on data redistribution than the other two approaches in the presence of data skews.

*3) Discussion:* From above results, it can be seen that two extra techniques, which are out the scope of this work, are also very critical for join implementations, in terms of reducing network traffic and communication time. Namely, efficient strategies on data partitioning for non-skew tuples and efficient approaches on identifying tuples which do not take part in a join. Because of the lack of these designs, we have not achieved an ideal result as we expected. Namely, our approach should have obviously advantages on data

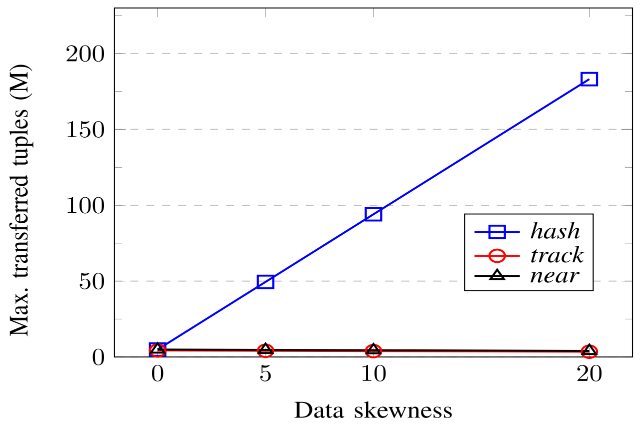

Figure 7. The maximum number of received tuples for each algorithm over different skews (over 200 nodes).

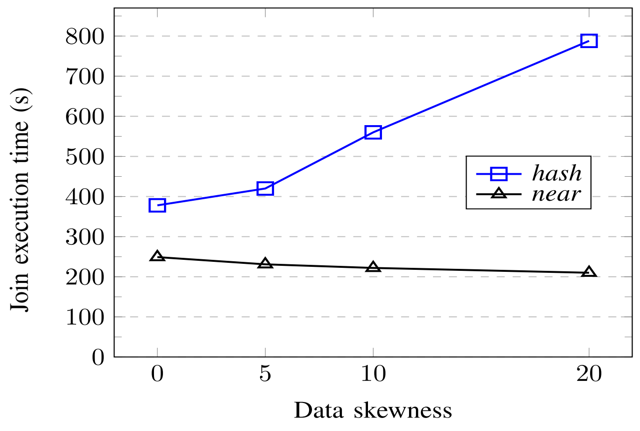

Figure 8. Comparison of join execution time over different data skews (with 200 nodes).

locality than neo-join and performs better on maximum transferred tuples than track-join. Nevertheless, we can see that the scheduling algorithm of our approach is much more lightweight than the state-of-the-art techniques, in the presence of large data and systems. Especially, our tests have shown that neo-join will not be able to be applied in large data systems. Moreover, though track-join can minimize network traffic and thus achieve very high data locality in various conditions, its network communication time is shown to be very similar to our approach. Considering their time cost on scheduling, we believe that our algorithm could be a better choice than track-join, in terms of join runtime efficiency in large-scale distributed scenarios. Additionally, though hash-join performs very well over the uniform distributed data, it is not robust in the presence of data skews.

### D. *Performance of Join Executions*

We compare the join performance of our approach against the most popularly used hash-join. As shown in Figure 8, we report the results over 200 nodes with different skews. It can bee seen that our approach can obviously outperform hash-join on join executions. Moreover, the runtime of our approach decreases slightly while that of hash-join increases greatly when increasing the data skew. This is consistent with the results of data locality and the maximum transferred tuples as described above and shows again the efficiency of our scheduling algorithm.

## V. Relate Work

As an important operation in large-scale data analytics applications, parallel joins can incur significant costs and hence improving efficiency of this operation have a significant impacts on overall performance of data queries. The two most conventional approaches in this filed is the hash-based and duplication-based join [13]. Though the hash-based scheme can achieve a near linear speedup under ideal balancing conditions [13], it has performance bottlenecks in the presence of data skew due to node hot spots [12]. In the meantime, as broadcasting operations always incur heavy communication cost, the duplication-based method is seldom adopted in large data joins, except for some work on its variants, which highly rely on underlying high-speed networks [14].

To improve the performance of join executions, various advanced algorithms over distributed architectures have been proposed [3], [5], [11], [15]. Regardless, almost all of them just focus on data skew handling and thus can only provide a coarse-grained operation over data redistribution. For example, the PRPD algorithm [3] just simply keeps all skewed tuples locally to reduce the network traffic. Moreover, though the statistics-based method [15] can generate efficient redistribution plans through construction of local and global histograms, it focuses on exploring the join relationships between input tuples instead of their data locality.

To minimize network traffic to speedup parallel joins, the state-of-the-art track-join approach [1] has proposed a more fine-grained operation, migration and select broadcast, based on the complete knowledge of the underlying data. Different from a conventional statistics approach, track-join records not only the frequency a key appears but also its location in the system. However, this could bring heavy overheads in the scheduling process of the approach. On the contrary, we just apply this kind of techniques to skewed tuples, which makes our method able to efficiently reduce network traffic, more importantly, much more lightweight. As we have shown in our experiments, though our algorithm has more network traffic than track-join, their numbers on maximum transferred tuples are very similar. Moreover, our scheduling algorithm is much faster than track-join, and can always be done in seconds.

As another the state-of-the-art approach, neo-join [7] proposes an optimal way on data assignment with targets to minimize the communication time. Different from a general redistribute approach, neo-join exhibits a *fuzzy* co-location of the partitioned data and adopts an optimal communication scheduling in their implementations. However, as we have analyzed, their optimization is based on solving an NP-complete problem, which is always expensive and thus cannot be applied to large-scale data systems. In comparison, our approach performs very fast and we can achieve similar assignment solution as [7] when the number of nodes is small as we have shown in our experiments.

Many other techniques, such as DHT [16] and dynamical scheduling [17], [18], have been proposed to improve performance of join operations. For example, the approach presented in [17] divides the join workload into fine-grained computation tasks and then scheduling them dynamically at runtime to avoid idle nodes and thus fully utilizes the computing power of a cluster. However, these work focus on join implementations in a *distributed* way, which is different from the parallel problem we have studied in this work.

## VI. Conclusion and Future Work

In this work, we have introduced a new approach, near-join, for efficient parallel joins over distributed systems. The approach has been devised specifically to efficiently reduce both network traffic and communication time during data redistribution of a join. Our experiments demonstrate that our scheduling algorithm is much more lightweight compared to the state-of-the-art approaches and our join implementation can achieve obvious speedups compared to the conventional hash-join approach.

Though our current implementation can efficiently explore the data locality for skewed tuples, it shows only a little improvement on the non-skewed ones in this aspect (also happens on the neo-join). As we have explained, the reason could be the data partitioning method we have used. We will try to adopt more advanced techniques to improve this problem in our future work (e.g., using radix partitioning [19]). In the meantime, to avoid the redistribution of tuples which do not participate a join, we will also consider to integrate a lightweight approach such as bloom-filter [20] in our implementation, to guarantee that our schedule still remains lightweight. We believe that all of these could further reduce communication time and improve join performance of our approach.

**Acknowledgments.** This work was partially supported by the German Research Foundation (DFG) within the Cluster of Excellence "Center for Advancing Electronics Dresden" (cfaed), the Collaborative Research Center SFB 912 (HAEC) and Emmy Noether grant KR 4381/1-1 (DIAMOND). The computations were performed on the Bull HPC Cluster at the Center for Information Services and High Performance Computing (ZIH) at TU Dresden.

## References

[1] O. Polychroniou, R. Sen, and K. A. Ross, "Track join: distributed joins with minimal network traffic," in *SIGMOD*, 2014, pp. 1483–1494.

[2] C. Kim, T. Kaldewey, V. W. Lee, E. Sedlar, A. D. Nguyen, N. Satish, J. Chhugani, A. Di Blas, and P. Dubey, "Sort vs. hash revisited: Fast join implementation on modern multi-core CPUs," *PVLDB*, vol. 2, no. 2, pp. 1378–1389, 2009.

[3] Y. Xu, P. Kostamaa, X. Zhou, and L. Chen, "Handling data skew in parallel joins in shared-nothing systems," in *SIGMOD*, 2008, pp. 1043–1052.

[4] L. Cheng, S. Kotoulas, T. E. Ward, and G. Theodoropoulos, "Robust and skew-resistant parallel joins in shared-nothing systems," in *CIKM*, 2014, pp. 1399–1408.

[5] N. Bruno, Y. Kwon, and M.-C. Wu, "Advanced join strategies for large-scale distributed computation," *PVLDB*, vol. 7, no. 13, 2014.

[6] S. Bellamkonda, H.-G. Li, U. Jagtap, Y. Zhu, V. Liang, and T. Cruanes, "Adaptive and big data scale parallel execution in Oracle," *PVLDB*, vol. 6, no. 11, pp. 1102–1113, 2013.

[7] W. Rödiger, T. Muhlbauer, P. Unterbrunner, A. Reiser, A. Kemper, and T. Neumann, "Locality-sensitive operators for parallel main-memory database clusters," in *ICDE*, 2014, pp. 592–603.

[8] T. Gonzalez and S. Sahni, "Open shop scheduling to minimize finish time," *Journal of the ACM*, vol. 23, no. 4, pp. 665–679, 1976.

[9] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. Von Praun, and V. Sarkar, "X10: an object-oriented approach to non-uniform cluster computing," *ACM SIGPLAN Notices*, vol. 40, no. 10, pp. 519–538, 2005.

[10] TPC-H benchmark specification, *http://www.tpc.org/tpch/*.

[11] W. Liao, T. Wang, H. Li, D. Yang, Z. Qiu, and K. Lei, "An adaptive skew insensitive join algorithm for large scale data analytics," in *APWeb*, 2014, pp. 494–502.

[12] D. J. DeWitt, J. F. Naughton, D. A. Schneider, and S. Seshadri, "Practical skew handling in parallel joins," in *VLDB*, 1992, pp. 27–40.

[13] D. DeWitt and J. Gray, "Parallel database systems: the future of high performance database systems," *Commun. ACM*, vol. 35, no. 6, pp. 85–98, Jun. 1992.

[14] P. W. Frey, R. Goncalves, M. Kersten, and J. Teubner, "Spinning relations: high-speed networks for distributed join processing," in *DaMoN*, 2009, pp. 27–33.

[15] M. Al Hajj Hassan and M. Bamha, "An efficient parallel algorithm for evaluating join queries on heterogeneous distributed systems," in *HiPC*, 2009, pp. 350–358.

[16] G. S. Manku, "Routing networks for distributed hash tables," in *PODC*, 2003, pp. 133–142.

[17] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia, "Skewtune: mitigating skew in MapReduce applications," in *SIGMOD*. ACM, 2012, pp. 25–36.

[18] X. Zhang, T. Kurc, T. Pan, U. Catalyurek, S. Narayanan, P. Wyckoff, and J. Saltz, "Strategies for using additional resources in parallel hash-based join algorithms," in *HPDC*, 2004, pp. 4–13.

[19] S. Manegold, P. Boncz, and M. Kersten, "Optimizing main-memory join on modern hardware," *TKDE*, vol. 14, no. 4, pp. 709–730, 2002.

[20] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, 1970.